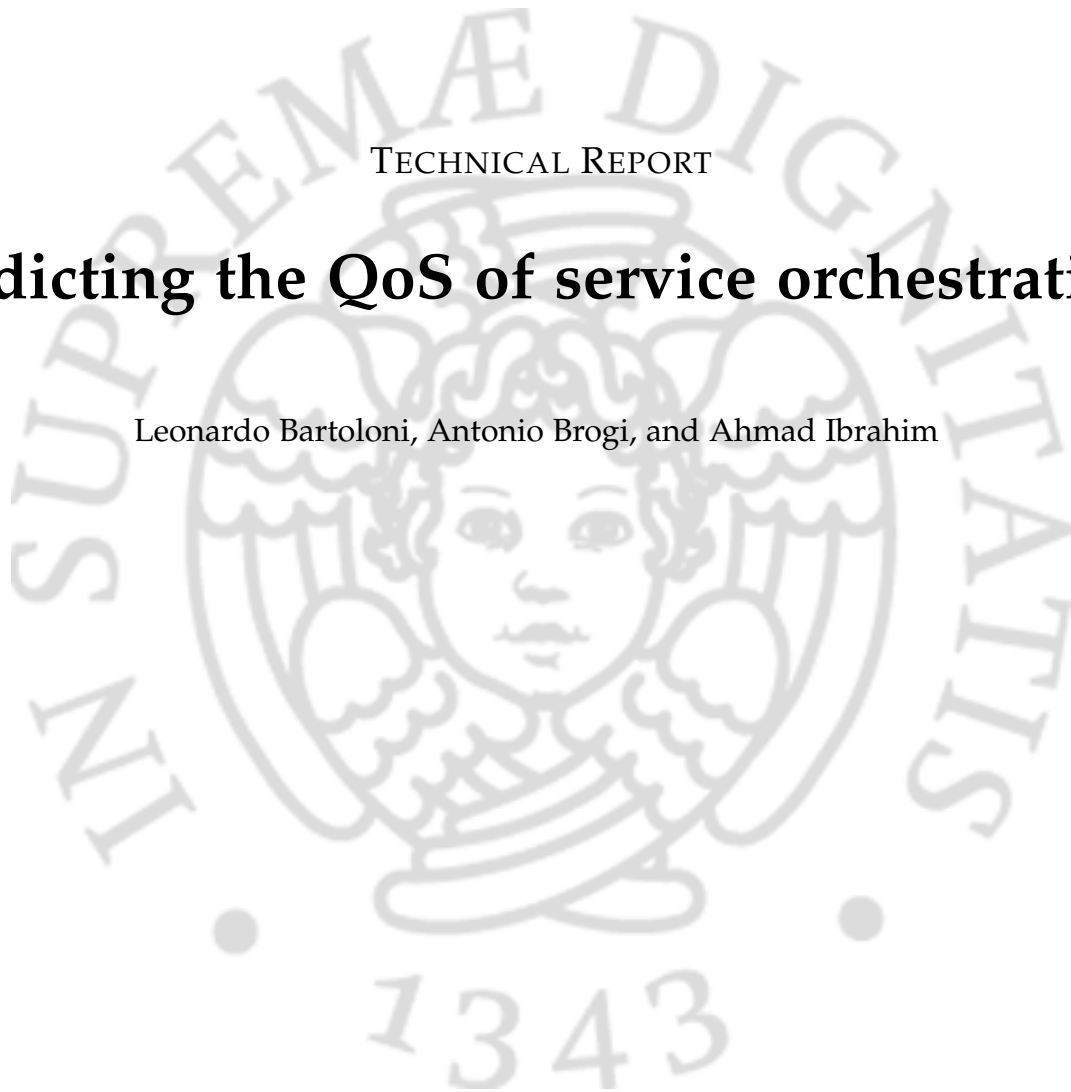


UNIVERSITÀ DI PISA
DIPARTIMENTO DI INFORMATICA

TECHNICAL REPORT

Predicting the QoS of service orchestrations

Leonardo Bartoloni, Antonio Brogi, and Ahmad Ibrahim



March 10, 2015

LICENSE: Creative Commons: Attribution-Noncommercial - No Derivative Works

ADDRESS: Largo B. Pontecorvo 3, 56127 Pisa, Italy. TEL: +39 050 2212700 FAX: +39 050 2212726

Predicting the QoS of service orchestrations

Leonardo Bartoloni, Antonio Brogi, and Ahmad Ibrahim

Abstract—The ability to a priori predict the QoS of a service orchestration is of pivotal importance both for the design of service compositions and for the definition of their SLAs. In this paper we present an algorithm to probabilistically predict the QoS of WS-BPEL service orchestrations. Our algorithm employs Monte Carlo simulations and it improves previous approaches by coping with complex dependency structures, unbound loops, fault handling, and unresponded service invocations. A proof-of-concept implementation of the algorithm in F# is described.

Index Terms—M.12.0.d Probabilistic reasoning, M.8.3 a Orchestration and Workflows, M.11.0.b QoS.



1 INTRODUCTION

SERVICE-ORIENTED computing [2] enables a rapid and low cost development of applications by using functionalities provided by available Web services. To achieve a specific business process, multiple Web services are composed and invoked in a specific (partial) order. While creating such compositions, both functional and non-functional properties of services play an important role. The functional requirements of a service must match the functional requirements of the application to be developed since otherwise latter might fail to function properly. The non-functional properties of services should also be considered since otherwise the application may not achieve the desired QoS level.

Quality of Service (QoS) [3] refers to a set of non-functional attributes used to describe service quality. More precisely, QoS refers to the ability of a Web service to respond to invocations according to the mutual expectations of both its provider and its customers [2]. Typical examples of QoS properties are latency, response time or reliability [3]. QoS is a key concept in service-oriented computing, and in service composition in particular. A service provider can advertise different levels of QoS to different customers with different prices, while customers can select appropriate offers according to their needs.

It is important to observe that the QoS of a service orchestration *does* depend on the QoS of the services it invokes. And the actual QoS featured by an invoked service depends on various conditions (e.g., network, server workload, and so on). Moreover, since complex applications can invoke multiple external services, a QoS change in one or more of those external services can seriously impact on the performance of the whole application. Hence, when selecting and composing various services together, the designer of an orchestrator should evaluate whether the obtained composition will yield an overall QoS level which is acceptable for the application.

One (obvious) way to estimate the QoS of a service composition is to deploy it on some infrastructure and to

measure QoS parameters over a sufficiently high number of executions. Unfortunately such an approach may be expensive both in time and in monetary cost (if non-free services are invoked), and it may also be not effective if some service invocations have side-effects. On the other hand, predicting the QoS of service orchestration is challenging, mainly because of four characteristics of service orchestrations.

- *Different results of service invocations.* Each invoked service can return a successful reply, a fault notification, or even no reply at all. If a fault is returned, a fault handling routine will be executed instead of the normal control flow. If no reply is received, the orchestrator may wait forever for a reply (unless some parallel branch throws a fault). In either case, the resulting QoS of the composition differs from the case of successful invocation.
- *Non-determinism in the workflow.* Different runs of the same application can yield different QoS values just because the orchestration control flow is non-deterministic due to two reasons. Firstly, different runs of the orchestration can get different service invocation results (success/fault/no reply). It is worth noting that a service is not always faulty or successful, rather it has a certain probability of being successful (as “guaranteed” in its SLA). Secondly, some control flow structures (alternatives and iterations) depend on input data which may differ in different runs. This may lead, for instance, to different numbers of iterations or to different branches executed in alternatives. Moreover certain QoS properties of invoked services can vary from one run to another (e.g., response time).
- *Global correlation.* The non-determinism in the workflow cannot be expressed correctly by probabilities of execution associated to each activity, because they may depend from each other in non obvious ways, for example, when their execution is subject to conditions on the same variables (we will illustrate a simple example of this in Sect. 2.).
- *Complex structures.* The control flow imposed by synchronization (that is, whenever a task needs to wait for another to complete before starting) on parallel

• L. Bartoloni, A. Brogi and A. Ibrahim are with the Department of Computer Science, University of Pisa, Italy.
E-mail: {bartolon, brogi, ahmad}@di.unipi.it

• This paper extends [1], where the authors provided a short and preliminary description of our results.

activities is more expressive than what is allowed by parallel execution only (with synchronization barriers at the end of each parallel task). This means that workflows which have such complex synchronization structures cannot be decomposed into parallel and sequential execution.

The objective of this paper is to present an algorithm to probabilistically predict the QoS of a workflow defining a service orchestration. The inputs of the algorithm are a WS-BPEL [4] workflow, and probability distributions for the QoS properties of the invoked services as well as for branch guard evaluations. The output of the algorithm is a probability distribution for the QoS properties of the orchestration.

We chose WS-BPEL because it is the OASIS standard for orchestrating web services and it features the aforementioned synchronization methods among parallel tasks (viz., synchronization <link>s within <flow> activities).

Our approach advances the state of the art in two main aspects. First our algorithm is able to handle workflows containing arbitrary dependency structures (i.e., not just parallel and sequential execution patterns), fault handling and unbound loops. This is obtained by employing different basic composition functions, instead of traditional sequential and parallel decomposition, which we show can more suitably model the language structures. For this reason we claim our approach to be compositional at language level. Our approach also features a more accurate management of correlations and hence more accurate results on some workflows with respect to previous approaches [5].

We represent distributions as *sampling functions*. The advantage of representing results as sampling functions is that not only we can compute average/expected values but also many other statistical properties (e.g., standard deviation or the probability of QoS not respecting a target SLA) by using the Monte Carlo method [6]. Moreover, as outputs and inputs are homogeneous, an output sampling function can be used as input for another computation (i.e., when an invoked service is itself an orchestration), making the whole approach compositional also at service level.

The rest of the paper is organized as follows. In Section 2 we discuss some of the challenges in probabilistically predicting the QoS of a service orchestration. In Section 3 we present our algorithm to determine the QoS of WS-BPEL service orchestrations, while in Section 4 we present a proof of concept implementation of our algorithm. In Section 5 we present two examples along with results of applying our algorithm to the examples. In Section 6 we comparatively discuss related work, while in Section 7 we draw some concluding remarks.

2 MOTIVATIONS

In order to determine the QoS of a service composition, different challenging aspects must be taken into account.

2.1 Non-determinism in the control flow.

The control flow of a workflow can contain conditions which usually depend upon the input data. Input data may vary from one execution to another, due to which the total

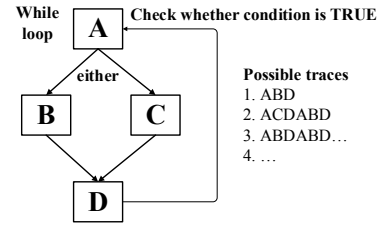


Fig. 1: Example of non-determinism in a workflow.

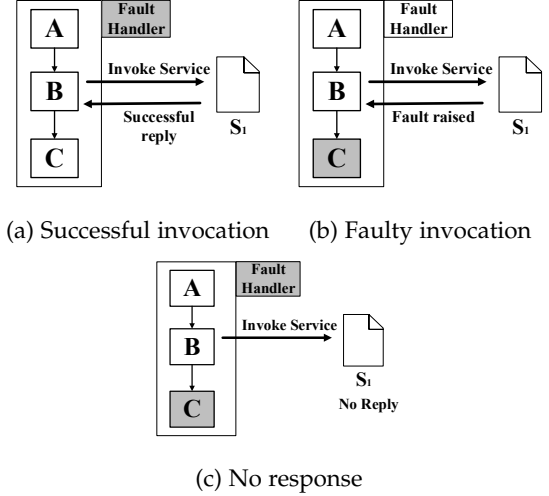


Fig. 2: Different results of service invocations.

number of iterations (in case of loop) and which branch will be executed (in case of if/else) is not known in advance.

For example, in the workflow depicted in Figure 1, the running time of the orchestration will depend on how many times the loop body will be executed and on which of the branches B and C will be taken each iteration (as one may take more time than the other).

2.2 Different results of service invocations.

Workflow can invoke external service to perform some computations. The result of such invocations can be successful, faulty or no reply.

For example, in the case of successful invocation, the workflow depicted in Figure 2 will continue normally, i.e., C will be executed after B (Figure 2a). In case of faulty invocation, a fault handler routine will instead run after B (Figure 2b). Moreover, in case of no response from S₁, the workflow will wait at B unless a timeout occur (Figure 2c).

2.3 Correlations in parallel branches.

The aforementioned challenges would suggest a probabilistic approach to the solution. It is important to observe that the naive solution of assigning independent probabilities to the activities (e.g., [5]) may result in incorrect result.

For example, in the workflow depicted in Figure 3a, if activity A will trigger, then either activity B or C will trigger with 50% probability, and we know that activity D will definitely be executed with 100% probability anyway.

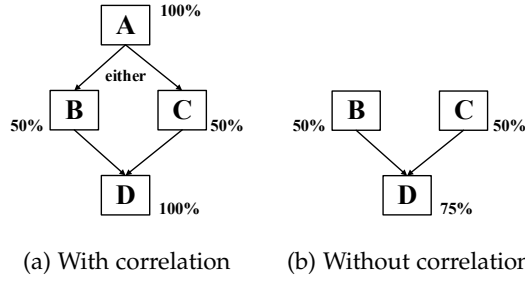


Fig. 3: Example of correlation in parallel branches.

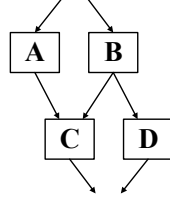


Fig. 4: Complex dependencies in a workflow.

However, if we ignore the information that activities B and C are correlated parallel branches which originated both from activity A, and just consider their probabilities of being executed (Figure 3b), then the probability that activity D will be executed is just 75%¹, which is incorrect.

2.4 Complex dependency structures.

Workflows can contain complex dependencies among activities, as per WS-BPEL synchronization `<link>s` [4]. Decomposing such workflows in terms of parallel and sequence constructs may not be always possible [7].

For example, Figure 4 shows a workflow that cannot be decomposed as sequential or parallel construct because of multiple incoming and outgoing links at activities.

In order to get accurate predictions, all the aforementioned problems must be suitably dealt with. As we will discuss in Section 6, to the best of our knowledge none of the previous approaches fully addresses all the aforementioned challenges.

3 OUR ALGORITHM

Our algorithm aims to provide a QoS estimate for a service orchestration based on the QoS of the services it invokes. To achieve that, we define a structurally recursive function that associates each WS-BPEL activity with a *cost* structure. Such a *cost* structure is a tuple of data from which it is possible to recover QoS values but it also contains enough extra information to compositionally determine the *cost* of structured activities defined by standard WS-BPEL constructs (e.g., *Sequence*, *Flow*, *IfThenElse*, *While*, *Scope*).

Please note that our scope is not to provide a specific tool to analyse WS-BPEL workflows but rather to describe an algorithm which can correctly analyse the QoS composition

in workflows exhibiting the four challenging aspects we identified in Section 2. For this reason we only consider a subset of WS-BPEL activities, namely *Invoke*, *Assign*, *Sequence*, *Flow*, *IfThenElse*, *While*, *Scope*.

In this section we define (Sec. 3.1) which is the minimal set of properties required for a *cost* data structure to be compositional w.r.t. the considered WS-BPEL language structures. We then show (Sec. 3.2) how a recursive evaluation function can be defined by exploiting these properties. Finally (Sec. 3.3) we show how we can introduce statistical non-determinism in the analysis and get average composition *costs* to predict the QoS of service orchestrations.

3.1 Cost compositors

As already anticipated, we define a *cost* structure that is compositional w.r.t. the WS-BPEL constructs. To this end, we associate a suitable composition function with each WS-BPEL activity. Intuitively speaking, we need to estimate what is the *cost* of, for instance, *Sequence*(A, B) as a function of the *costs* of A and B. Similarly, we need to be able to estimate the *cost* of *Flow*(activityList) as a function of the *costs* of each activity in activityList. While the *cost* composition function for *Sequence* is pretty straightforward, providing a *cost* composition function for generic *Flow* activities is more challenging.

In previous approaches (e.g., [8]), the *Flow* dependency graph is decomposed into parallel and sequence compositions, thus requiring only the two composition functions for parallel and sequential executions. However, this kind of decomposition can be done only for a limited subset of all possible dependency graphs (Section 2.4), so the language subset for which it is possible to compute QoS is significantly less expressive than what WS-BPEL's *Flow* construct allows [7].

In order to consider generic *Flows*, we define two different functions as base *cost* composition operations. The first base *cost* compositor is a parallel compositor: *Both*(A, B) is the *cost* associated with executing independently an activity with associated *cost* A and another one whose *cost* is B. The second function we need is what we call *Delay*. We use it to express the increase in *cost* of an activity when it needs to wait for another activity to complete before starting. *Delay*(A, B) is defined as is the cost A increased to take into account the time needed to wait for another activity (of cost B) to complete, hence the name. If an activity would cost A if it had no dependencies, but the workflow specified that it can be run only after another that costs B, then we need to consider its *delayed cost* *Delay*(A, B) when aggregating it with other activities in the workflow.

Note that function *Both* is commutative², *Both* and *Delay* are associative³, and *Delay* is right-distributive over *Both*⁴.

We also explicitly name a neutral element *Zero* (i.e., *Both*(A, *Zero*) = A and *Delay*(A, *Zero*) = A), which can be useful for example to define the *All* function, which extends the *Both* binary compositor to accept any number

1. The probability that neither B or C will be executed is $50\% \times 50\% = 25\%$. Thus the probability that D will be executed would be $100\% - 25\% = 75\%$.

2. $\forall a, b. \text{Both}(a, b) = \text{Both}(b, a)$.

3. $\forall a, b, c. \text{Both}(a, \text{Both}(b, c)) = \text{Both}(\text{Both}(a, b), c)$ and $\text{Delay}(a, \text{Delay}(b, c)) = \text{Delay}(\text{Delay}(a, b), c)$.

4. $\forall a, b, c. \text{Delay}(\text{Both}(a, b), c) = \text{Both}(\text{Delay}(a, c), \text{Delay}(b, c))$.

of parameters. Since we have prototyped our algorithm with F# [9], we use F# syntax to describe pseudo-code throughout the paper.

```
let rec All activityList =
    match activityList with
    | [] -> Zero
    | h:t -> Both(h , All t)
```

As an example we show the definition of `Both` and `Delay` for the two costs *expense* and *completion time* we considered to validate the approach (as we will see in Section 5).

For *expense*, i.e., monetary costs:

```
let Both( aExpense, bExpense ) =
    aExpense + bExpense
let Delay( aExpense, bExpense ) =
    aExpense
let All( expenses ) =
    Sum(expenses)
```

`Both` (and similarly its generalization, `All`) is the sum of costs, while `Delay` is just the original cost (since delaying an activity will not increase its expenses).

For *completion time*, i.e., the time required for an activity to complete:

```
let Both( aTime, bTime ) =
    Max(aTime, bTime)
let Delay( aTime, bTime ) =
    aTime + bTime
let All( times ) =
    Max(times)
```

`Both` is the maximum and `delay` is the sum of the two (because the delayed activity will start after the other has been completed).

3.2 Workflow analysis

In this section, we describe the pseudo-code implementing the probabilistic analysis of the WS-BPEL activities `Sequence`, `Scope`, `Flow`, `Assign`, `IfThenElse` and `While`. We have developed a parser which converts a WS-BPEL process into a F# term which consists only of `Both` and `Delay` constructs and which is recursively evaluated by the *exec* function, which is the structurally recursive evaluation function computing outcome and cost of WS-BPEL activities.

3.2.1 Environment and Outcome parameters.

For a realistic evaluation of a WS-BPEL activity it is necessary to take into account the results of the activities that have been previously evaluated. Such information can be stored in, and retrieved from, the *environment* and *outcome* parameters. WS-BPEL features two control-flow mechanisms that may lead to skipping the execution of some activities: “Explicit” control-flow activities (alternatives, iterations, synchronization `<link>s`), and fault management activities.

- “Explicit” control flow can be simply handled by evaluating branching conditions (viz., in `IfThenElse` and `While`) and transition conditions (viz., in synchronization `<link>s` within `Flows`). The *cost* depends also on the *environment* parameter of the *cost* evaluation function, which holds variable values over which condition are evaluated, as well as the status of synchronization `<link>s`.

- **Fault management** requires to keep track of the *outcome* of an activity, i.e., whether an activity was *successfully* executed or not. For instance, an activity that depends on a *faulty* activity will not be executed. Moreover, an activity can be *stuck* waiting for an event which will never occur (viz., if the invoked service will not return any reply). In this case any dependent activity or fault handler will not be executed.

Due to the importance of the *environment* and *outcome* parameters, we will explicitly include them in the following pseudo-code.

3.2.2 Sequence(a1,a2).

```
Sequence (a1,a2) ->
    let env1,outcome1,cost1 = exec env a1
    if outcome1 = Success then
        let env2,outcome2, cost2 = exec env1 a2
        env2, outcome2, Both(cost1,Delay(cost2,cost1))
    else
        env1,outcome1,cost1
```

The evaluation of `Sequence (a1, a2)` is straightforward:

- If the result `(env1,outcome1,cost1)` of evaluating `a1` is *successful* (viz., `outcome1=Success`), then `a2` is evaluated in the new environment `env1`. The environment (`env2`) and the result (`outcome2`) obtained by evaluating `a2` are then returned, together with the *cost* term `Both(cost1,Delay(cost2,cost1))`. As anticipated, the *cost* associated to `Sequence (a1, a2)` is the *cost* of executing `a1` and the *cost* of executing `a2` after `a1`.
- If the result `(env1,outcome1,cost1)` of evaluating `a1` is instead not *successful*, the such result is returned as the result of evaluating `Sequence (a1, a2)`.

3.2.3 Scope(a1,faultHandler).

```
Scope (a1,faultHandler) ->
    let env1,outcome1,cost1 = exec env a1
    if outcome1 = Fault then
        let env2,outcome2, cost2 = exec env1 faultHandler
        env2,outcome2,Both(cost1,Delay(cost2,cost1))
    else
        env1,outcome1,cost1
```

A `Scope` is associated with a `faultHandler` which is executed only if a *fault* occurs in activity `a1`. The evaluation of `Scope (a1, faultHandler)` is hence somewhat similar to the evaluation of `Sequence (a1, a2)`.

- If the result `(env1,outcome1,cost1)` of evaluating `a1` yields a *fault* (viz., `outcome1=Fault`), then `faultHandler` is evaluated in the new environment `env1`. The environment (`env2`) and the result (`outcome2`) obtained by evaluating `faultHandler` are then returned, together with the *cost* term `Both(cost1,Delay(cost2,cost1))`. Indeed, as anticipated, the *cost* associated to `Scope(a1, faultHandler)` is the *cost* of executing `a1` and the *cost* of executing `faultHandler` after `a1`.
- If the result `(env1,outcome1,cost1)` of evaluating `a1` is instead *successful* or *stuck*, then such result `(env1,outcome1,cost1)` is returned as the result of evaluating `Scope(a1,faultHandler)`.

3.2.4 Assign (name,expr).

```
Assign (name,expr) ->
  env.Add(name,boolExprEval env expr),Success,Zero
```

The Assign activity assigns (the result of evaluating) an expression to a variable *name* and adds it to the environment. We assume that the *outcome* of Assign is always *Success* and has *Zero* cost.

3.2.5 IfThenElse(guard,a1,a2).

```
IfThenElse(guard,a1,a2) ->
  let guardValue = boolExprEval env guard
  if guardValue then
    exec env a1
  else
    exec env a2
```

The IfThenElse activity first evaluates its guard condition:

- If it evaluates to *True*, then *exec* evaluates *a1*.
- If it evaluates to *False*, then *exec* evaluates *a2*.

3.2.6 While(guard,body).

```
While(guard,body) ->
  let guardValue = boolExprEval env guard
  if guardValue then
    exec env (Sequence (body,While(guard,body)))
  else
    env,Success,Zero
```

Similarly to IfThenElse, a While activity first evaluates its guard condition.

- If it evaluates to *False*, the body of the loop is skipped and *outcome* *Success* and *cost* *Zero* are returned.
- Otherwise, *exec* recursively evaluates the body along with While construct again in a Sequence.

3.2.7 Flow(activityList, linkList).

A Flow activity includes a set of activities (*activityList*) to be run in parallel subject to a (possibly empty) set of synchronization *<link>s* (*linkList*).

In WS-BPEL language the resolution of race condition is left unspecified, and may change according to the specific implementation of the WS-BPEL interpreter [4]. As a consequence, for some activities inside a Flow activity, it is impossible to determine whether they will be executed or not. For example if two activities A and B are part of a Flow block and A throws a fault, the activity B may or may not have been executed yet. In our approach we adopt a pessimistic outlook: we assume that all activities will be executed unless we can prove otherwise. This means that we will skip (i.e, consider them as having zero cost) an activity if it is the target of a synchronization *<link>* whose source is a faulty, stuck or likewise skipped activity, while unrelated activities are evaluated normally.

For each activity A directly inside the Flow we compute two quantities:

- The *outcome* of the activity, which depends also on the outcomes of its *preceedingActivities* (if any), that is of the activities which are source of the synchronization *<link>s* having activity A as target.
- The *delayedCost* of the activity, which is the cost of the activity itself delayed by the cost of all its *preceedingActivities* (if any).

To compute these quantities, we first pick a topological sort [10] of the *activityList* (i.e., any permutation of

activityList in which if activity A precedes activity B, then A is not the target of a *<link>* from B). For the purpose of environment updates, we can assume that activities are executed serially in that order, since the result is not affected by the chosen topological sort. The topological sort ensures that when evaluating an activity the required *delayedCost* and *outcome* have been computed for all *preceedingActivities* (so that they are not computed more than once).

When *delayedCost* and *outcome* have been computed for all activities, the algorithm computes cost and outcome for the Flow activity itself. If any of the activity is faulty or stuck, then the outcome of the Flow activity will respectively be *Fault* or *Stuck* (*Fault* has precedence over *Stuck* since a *Stuck* activity can be interrupted by a *Fault* from a parallel branch). We skip the evaluation all those activities whose *joinCondition* evaluates to false⁵.

The cost of the Flow activity itself is finally computed by putting together with the Both compositor the *delayedCost* of all activities inside it.

```
Flow (activityList, linkList) ->
  let mutable env = env
  let activityList = topoSort activityList linkList
  let delayedCosts = new Dictionary<string, Cost>
  let activityOutcomes = new Dictionary<string, Outcome>
  for activity in activityList do
    let incomingLinks =
      [ 1 in linkList where 1.target = activity.name ]
    let outgoingLinks =
      [ 1 in linkList where 1.source = activity.name ]
    let mutable preceedingActivities = []
    for 1 in incomingLinks do
      for a in activityList do
        if 1.source = a.name then
          preceedingActivities <- a::preceedingActivities
    let mutable outcomei = Success
    for a in preceedingActivities
      if outcomei = Success ||
        activityOutcomes.[a.name] = Fault then
        outcomei <- activityOutcomes.[a.name]
    let mutable costi = Zero
    for a in preceedingActivities do
      costi <- Both ( costi, delayedCosts.[a.name] )
    let joinCondition =
      evaluateBooleanExpression a.joinCondition env
    match outcomei, joinCondition, SuppressJoinFailure with
    | Success,true,_ ->
      let e,outcome,cost = exec activity env
      delayedCosts.insert(activity.name,Delay(cost,costi))
      activityOutcomes.insert(activity.name,outcome)
      env <- e
    | Success,false,true ->
      delayedCosts.insert(activity.name,Delay(Zero,costi))
      activityOutcomes.insert(activity.name,Success)
    | Success,false,false ->
      delayedCosts.insert(activity.name,Delay(Zero,costi))
      activityOutcomes.insert(activity.name,Fault)
    | other,_,_ ->
      delayedCosts.insert(activity.name,Delay(Zero,costi))
      activityOutcomes.insert(activity.name,other)
  for 1 in outgoingLinks do
    let linkStatus =
      evaluateBooleanExpression 1.transitionCondition env
    env <- env.Add ( 1.name, linkStatus )
```

3.3 Statistical non-determinism

In the previous section, for the sake of simplicity, we have assumed to be able to evaluate all conditions' values and to know whether an activity is *faulty* or not. Unfortunately, this is not possible in general. The evaluation of conditions

5. In this case if the *suppressJoinFailure* flag is set to true the activity's *outcome* is considered a *Success*, otherwise it is a *Fault*.

(usually) depends on data values, which are unknown a priori, and invocation results can vary too from one execution to another. In this section, we describe how Monte Carlo simulations can be employed to address the analysis of correlations, non-determinism and different invocation results (mentioned in Section 2). We also present the pseudo-code for the `Invoke` and `OpaqueAssign` activities.

3.3.1 Monte Carlo simulation.

Monte Carlo simulation [6] is a technique that employs repeated random sampling to obtain numerical results. Generally, simulation is run multiple times in order to obtain the distribution of a probabilistic entity. We can compute an estimation of expected values (i.e., a probability weighted average of a function applied to all possible values) for many quantities by averaging the results of different iterations.

Monte Carlo simulation is useful for our algorithm in two ways. First, at each iteration of Monte Carlo we can sample the conditions of *branches* and *loops* (by using the sampling function) and deterministically decide what to execute. This, along with recursive sampling, allows us to address correlations, non-determinism and different invocation results. Second, many QoS properties can be written as expectation queries. For instance, *reliability* is the expected value of a sampling function that returns 1 if the *outcome* is *success*, and 0 otherwise. *Average cost and time* are the expected value of the cost and time for the entire workflow respectively. While the Monte Carlo simulation will give an approximated result, it is possible to improve accuracy arbitrarily by increasing the number of samples. The generation of samples is independent of previous iterations, and thus it can be run in parallel.

3.3.2 Sampling Functions.

A sampling function for a certain probability distribution is an algorithm which generates samples according to such a distribution. In a Monte Carlo simulation sampling functions are required to generate samples for random variables (i.e., variables which are assigned values randomly according to a certain distribution). In the context of WS-BPEL workflow analysis this is needed for the evaluation of opaque expressions as well as of *outcome* and *cost* for `Invoke` activities.

If we assume that the probability distributions of variable values and of invocation *outcomes* and *costs* (whatever definition of *cost* is employed to evaluate QoS properties we are interested in) are known, then we can estimate the probability of an invocation throwing a *fault* or getting *stuck*, or the probability of a variable being true or false when evaluating an expression, as well as the total *cost* for the composition. By exploiting this information over distributions we can write a sampling function which generates each time a new sample for the value of the variable assignment (i.e., when evaluating assignment with non-deterministic expressions) or invocation *outcome/cost* (i.e., when evaluating `Invoke` activities), each time picking one of the possible values with its probability. The sampling function for `Invoke` can generate samples for *outcome* in the form of *Success*, *Fault* and *Stuck*. On the other hand, the sampling functions for condition evaluation can generate

samples in the form of true and false values which can help in branch selection and loop continuation.

As an example, we show below a pseudocode for a sampling function for a Bernoulli distributed variable (that is, a boolean variable true with probability *p*, which we can use for variable assignment) by exploiting a pseudo-random number generator. This can be similarly extended to create sampling functions for more complex domains (e.g., `Invoke`). In our implementation we use a similar algorithm to generate a sampling function according to a list of possible values with associated probability that is specified in the input, in the `Annotations.xml` file (please see Section 4 for details).

```
let bernoulli p () =
  if generator.NextDouble() < p then
    true
  else
    false
SamplingFunCondition.[variableName]<-bernoulli probability
```

In order to preserve correlations when implementing Monte Carlo algorithms it is required to take care not to sample the same random variable twice during a single simulation run (that is, during a simulation run a variable should have a fixed value and not a different value each time it is evaluated). This is one of the reasons for which we store computed values for activities in a `Flow` instead of evaluating them when necessary. We only allow statistical non-determinism in two constructs of the language: `Invoke` and `OpaqueAssign`. For these activities the evaluation consists of just running the respective sampling functions, as we show below.

3.3.3 Invoke (partnerLink).

```
Invoke (partnerLink) ->
  let outcome,cost = SamplingFunInvoke.[partnerLink] ()
  outcome,cost,env
```

The `Invoke` activity is used to call an external service via a `partnerLink`⁶. We associate each endpoint with a sampling function. To evaluate the `Invoke` activity, the algorithm will retrieve the sampling function for the associated endpoint from a dictionary, using its `partnerLink`, then execute it to generate a sample of *outcome* and *cost*, and return it. The environment is left unchanged.

3.3.4 OpaqueAssign (variableName).

```
OpaqueAssign (variableName) ->
  let value = SamplingFunCondition.[variableName] ()
  Success,Zero,env.Add(variable,value)
```

WS-BPEL permits to specify hidden (opaque) assignments. An opaque expression is a placeholder for a corresponding executable WS-BPEL expression. The `OpaqueAssign` activity that we consider represents a variable assignment using an opaque expression.

The evaluation of `OpaqueAssign` initially calls the sampling function to generate a (true/false) sample, then it updates the environment by assigning the returned value to the variable. This activity has always a *Success outcome* and *Zero cost*.

We assume that hidden expression are only used in assignments to variables instead of supporting generic usage

6. For the sake of simplicity, we assume that different endpoints can be identified by different `partnerLink` names.

of them whenever WS-BPEL would allow it (for example, in branches or loop guards) because those cases can always be rewritten as a sequence of an `OpaqueAssign` to a temporary variable followed by a deterministic `IfThenElse/While` having that variable as guard⁷. For example

```
<if>
  <condition>some-opaque-expression</condition>
  activity
<else>
  otherActivity
</else>
</if>
```

can be rewritten as

```
<sequence>
  <assign>
    <copy>
      <from>some-opaque-expression</from>
      <to>temp-variable</to>
    </copy>
  </assign>
  <if>
    <condition>temp-variable</condition>
    activity
  <else>
    otherActivity
  </else>
</if>
</sequence>
```

A sample implementation of a Monte Carlo method follows:

```
let costExpectation iterations projection workflow =
  let mutable sum = 0.0
  for i = 0 to iterations do
    let env = Map.empty
    let outcome, cost = Eval workflowDefinition env
    sum <- sum + (projection (outcome, cost))
  sum / iterations
```

where `iterations` is the number of samples used (the higher the number the higher the precision), `projection` is the function that extracts from `cost` and `outcome` the value of which we want to compute the expectation value, and `workflow` is the WS-BPEL workflow we are analyzing.

In the sample implementation (Section 4) we are creating sampling functions from a list of various possible values and their probabilities. The algorithm however works with any procedure which allows to generate samples for the endpoint `outcome` and `cost`. A particularly interesting case is when an endpoint is described by another WS-BPEL workflow, for which the sample evaluation itself (that is, the `exec` function applied to the description of the external service) may be used as external sampling function for that service's invocation in the primary workflow (Figure 5). This, together to the fact that activity `cost` and `outcome` may be computed from the `cost` and `outcome` of activities inside it, is the reason for which we qualify our approach as “truly compositional”.

Keeping samples for activities instead of probabilities allows us to preserve correlation during the same Monte Carlo iteration (and the environment ensures variable values are evaluated only once for each iteration).

4 PROOF-OF-CONCEPT IMPLEMENTATION

We developed a proof of concept implementation of our algorithm (Section 3) in the PASO (Probabilistic Analyser

7. Indeed in our implementation we allow opaque expressions in `IfThenElse` and `While` by transforming them into sequences during the parsing.

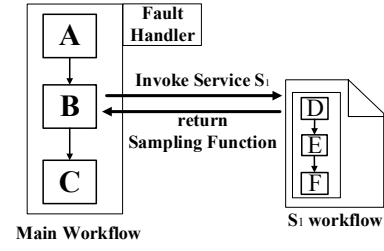


Fig. 5: External sampling function for service invocation.

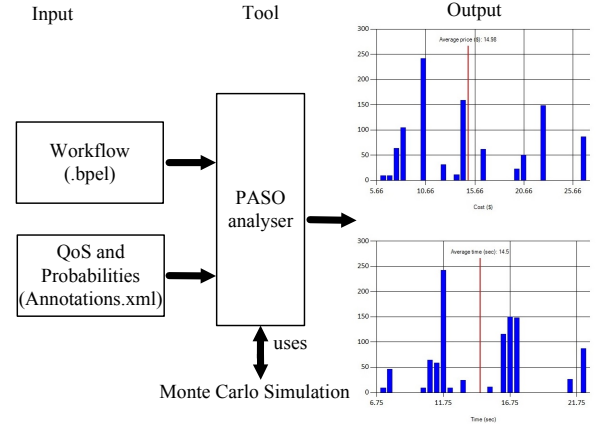


Fig. 6: Bird-eye view of the input-output behaviour of the PASO analyser.

of Service Orchestration) tool, which is a desktop application developed using F# .Net [9]⁸. Although the proposed algorithm can be implemented in any language, we chose F# because it is a strongly typed language and it allows fast and convenient ways to prototype inference rules.

The analyser tool takes two inputs: a WS-BPEL process, and QoS and probability distributions. Using these inputs, the PASO analyser parses the WS-BPEL process and applies Monte Carlo simulation to generate the output in the form of desired QoS properties (e.g., response time, cost, reliability, and so on), as sketched in Figure 6.

We now describe the format of the input, output and the internal functioning of the PASO analyser.

4.1 Analyser input

As stated previously, the first input is a WS-BPEL process (a .bpel file) while the second input is QoS and probability distributions represented in an `Annotations.xml` file.

4.1.1 WS-BPEL workflow.

PASO is able to analyse a subset of WS-BPEL structural (Sequence, Flow, `IfThenElse`, `While` and `Scope`) and basic activities (`Invoke`, and `Assign`). Other basic activities, such as `Receive`, `Reply` are ignored, and considered with zero cost and always successfully executing.

8. The source code for the PASO analyser and the library can be downloaded from <https://github.com/upi-bpel/paso/tree/ieee-full>

4.1.2 QoS and Probability distribution.

To estimate the QoS of an orchestration, we need information about QoS of individual services used in the orchestration (e.g., cost and response time). Moreover, in order to resolve control flow in our Monte Carlo simulations, we also need information about the *outcome* distribution of invoked services.

We thus require the user to provide annotations of probabilities for outcomes and costs of service invocations, and the probability of truth for the conditions used in `IfThenElse` and `While` activities. From these probabilities⁹ we internally construct sampling functions to use during Monte Carlo simulation (see Bernoulli Sampling function in Section 3.3.2).

While there are various proposals of how to represent QoS (e.g., WSLA [11], WS-Agreement [12] or WS-Policy [13]), a simple XML structure to represent QoS and probability distributions was sufficient for our purposes. We now describe how to represent input in Annotations.xml.

The following annotation shows a sample condition which annotates an opaque condition with its probability being true¹⁰.

```
<condition>
  <name>LoanRequest &gt; 10000$</name>
  <value probability="0.5" >True</value>
</condition>
```

When the workflow contains a condition which is not annotated, we assume it to be a deterministic condition, that is a condition which can be evaluated from the variables already assigned¹¹.

Similarly, the following annotation shows a sample that will be used to construct a sampling function for an `Invoke` endpoint¹². The annotation lists the possible outcomes of sampling (success, fault or stuck) along with their probabilities and associated costs. If an outcome may correspond to different costs it can be repeated: the total probability for the outcome will be the sum of probabilities. If the sum of all probabilities for an endpoint is not equal to 1 the probabilities will be renormalized.

```
<endpoint>
  <name>riskAssessor</name>
  <partnerLink>assessor</partnerLink>
  <event outcome="success" probability="0.79">
    <cost> 0.1$</cost>
    <time> 1 sec</time>
  </event>
  <event outcome="success" probability="0.2">
    <cost> 0.1$</cost>
    <time> 2 sec</time>
  </event>
  <event outcome="fault" probability="0.0">
    <cost> 0$</cost>
    <time> 0 sec</time>
  </event>
  <event outcome="stuck" probability="0.01">
    <cost> 0.1$</cost>
    <time> 0 sec</time>
  </event>
```

9. These probabilities may be deduced from Service Level Agreements (SLAs), or statistically inferred from data such as logs or performance counters if available.

10. We use as key the string expressing the condition itself, assuming it to be unique in a process, and we expect probability to be expressed as a floating number between 0 and 1.

11. PASO supports a minimal Boolean expression language in order to specify such conditions.

12. We assume that the name of `partnerLink` in `Invoke` is unique in a process.

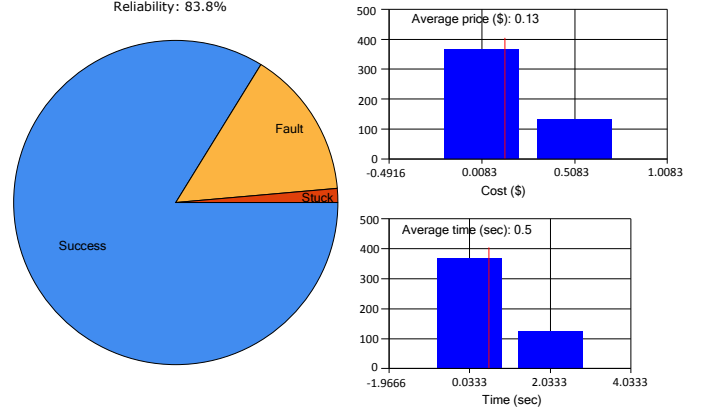


Fig. 7: An example output generated by PASO.

```
</event>
</endpoint>
```

4.2 Analyser output

The analyser tool executes the algorithm multiple times over the same inputs (due to Monte Carlo simulation). The output at the end is a histogram and pie chart summarizing both individual and average values for Cost, Response time and Reliability (Figure 7).

4.3 Internal design of the analyser tool

The proposed algorithm is implemented in the form of a library and an executable. The front-end executable parses WS-BPEL file and the annotation syntax, constructing the abstract syntax tree of the workflow and the sampling functions from the annotations. These are passed to the back-end library, which implements the *exec* function which computes cost and outcome for the workflow, and the Monte Carlo algorithm. The resulting data are then passed back to the executable, which displays them as graphs.

Support for other similar workflow languages with different syntax than WS-BPEL can be easily implemented by modifying only the front-end parser (for example, a bigger subset of WS-BPEL including different loops and *elseif* constructs).

In our front-end we wrote two different parsers, one for WS-BPEL language and one for the minimal Boolean expression language that we allow to be used in conditions. The abstract syntax tree that we used for representing workflow actually differs from WS-BPEL XML node nesting (mainly in the flow `<link>` list whose source and target are not specified in the link itself but inside the activity).

5 EXAMPLES

To illustrate our approach, we will apply our algorithm to two examples: A loan request and a shipping service. Both of them are well-known examples adapted from [4]. We will try to demonstrate the usefulness of our algorithm with the help of these examples. The first one covers a non-deterministic scenario, different invocation outcomes and complex dependencies, while the second illustrates

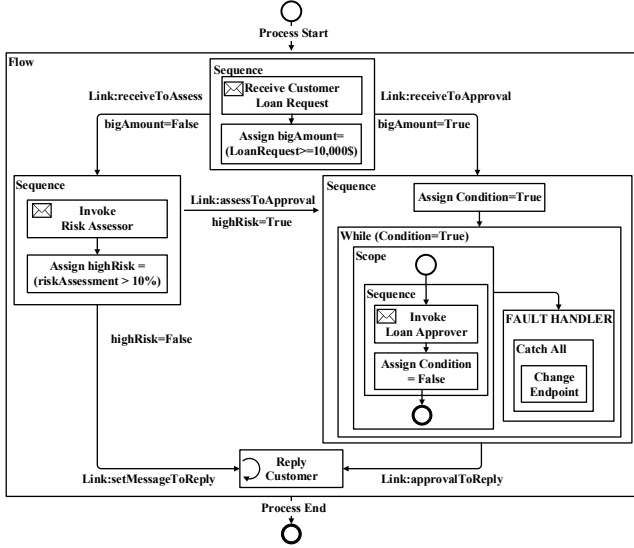


Fig. 8: Loan request example.

correlations in parallel branches. In this section we will describe these two examples and then show the result of the assessment.

For these examples, we want to estimate values for the following QoS properties of this composition:

- *Reliability* [3], [14], as the probability of an execution to be *successful*. For a composition run to be successful we require it to complete before the SLA guaranteed response time. We assume guaranteed response time of 40 minutes for loan example and of 15 seconds for shipping service example.
- *Amortized expense for successful execution* [3], [14], [15], the average expense for reaching a successful execution (which also includes the cost for unsuccessful attempts). For example, if we have 10 invocations, each one with an average 2\$ expense and only 8 of them are successful then the amortized expense for successful execution would be $\frac{2\$ \cdot 10}{8} = 2.5\$$.
- *Average response time* [3], [14], [15], computed only for successful executions.

5.1 Loan request example

The first example is the bank customer loan request example (Figure 8). The customer sends a request for a loan to the bank. This request will contain the loan amount as well as some personal data on the customer. Upon receiving the request, the system will check whether the requested amount is less than 10,000\$ or not. If the request is less than 10,000\$, then the system will invoke a risk assessor service to assess the risk and get a risk evaluation (an estimated probability of the client not being able to repay the loan). If the risk is low (below 10%), then the loan request will be approved automatically and a reply will be send to customer. Otherwise, the request will be forwarded to a loan approver service (a human accountant). In case of successful execution of the loan approver service, the system will reply to the customer. In case of fault it will bind the loan approver service to a different endpoint (another accountant) and it

TABLE 1: Input distributions for the loan request example.

	True	False
LoanRequest \geq 10,000\$	50%	50%
riskAssessment > 10%	60%	40%

(a) Control Flow

	Success	Fault	Stuck
0.1\$, 1 sec	79%	-	-
0.1\$, 2 sec	20%	-	-
0.1\$, 0 sec	-	-	1%

(b) Risk Assessment

	Success	Fault	Stuck
5\$, 10 min	30%	-	-
10\$, 20 min	35%	-	-
15\$, 30 min	20%	-	-
0\$, 5 min	-	15%	-

(c) Approval

will invoke the service again, until the service is executed successfully.

Let us assume the following distribution of conditions and invoked services QoS (summarized in Table 1):

- The condition *LoanRequest* \geq 10,000\$ is true 50% of times.
- The condition *riskAssessment* > 10% is true 60% of times.
- *risk assessor* invocation costs 0.1\$ per use. It usually (79%) completes in 1 second. Sometimes (20%) it will take 2 seconds because of congestion. In rare cases (1%) we do not receive a reply (stuck), but we will assume that the cost has already been paid.
- *loan approver* is bound to the availability of a human accountant. When the accountant is busy (15%), the invocation result is a fault and takes 5 minutes. Otherwise it takes a longer time and an expense which is proportional to it (the accountant hourly wage): 30% probability completing in 10 minutes with 5\$ expense, 35% probability completing in 20 minutes with 10\$ expense, and 20% probability completing in 30 minutes with 15\$ expense.

5.2 Shipping service example

The second example is the goods shipping service example (Figure 9). A shipping service receives a shipping order from the customer. A shipping order contains a list of requested items and shipment instructions along with other information. As per shipment instructions, an item can be shipped separately or combined with other items. Upon receiving an order, the shipping service initially checks shipment instructions. In case of individual shipment, each item is shipped separately, otherwise all items are shipped together. At the end, the customer is notified with a shipment complete message.

For the input, let us assume the following distribution of conditions and invoked service QoS (summarized in Table 2):

- The condition *ShipIndividual* in IF is True 70% of times.
- The condition *Item*<*TotalItems* in While loop is True 60% of times.
- Invoke service *ShipItem* would cost 0.5\$ with response time of 2 seconds. (Our aim is to highlight the

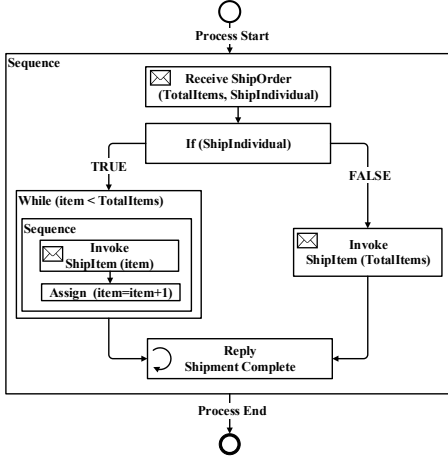


Fig. 9: Shipping service example.

TABLE 2: Input distributions for the shipping service example.

	True	False
ShipIndividual	70%	30%
Item<TotalItems	80%	20%

(a) Control Flow

	Success	Fault	Stuck
0.5\$, 2 sec	100%	-	-

(b) ShipItem

effect of correlation through this example. Since the previous Loan example already simulates the result of three outcomes for service invocation, we will not highlight this aspect again and will assume 100% success rate).

With the above two examples, a natural question is: what will (probably) be the QoS of the orchestration based on given input? We will illustrate next how our algorithm can be exploited to answer such question.

5.3 Results

Now we will apply our algorithm (Section 3) on the two examples to estimate the desired QoS properties, namely *average response time*, *amortized expense for successful execution*, and *reliability*. We illustrate step-by-step how the `exec` function can compute a sample for the orchestration's *cost* and *outcome* parameters. Obviously, as control flow depends on random values, the path we show is just one of the many possible execution traces. We then show how to perform a Monte Carlo sampling to estimate the required QoS properties. Finally we show the result of performing such sampling over a large number of samples.

To aggregate samples we use the following projection function:

```
let Projector(outcome, (expense, time)) =
  let successCount, successfulTime =
    if outcome = Success and time < timeLimit
    then 1.0, time
    else 0.0, 0.0
  (successfulTime, expense, successCount)
```

Reliability can be determined by computing the expectation of `successCount`. *Amortized expense* and *average*

response time are divided by *reliability* to normalize them with respect to the number of successful executions.

```
let (expectedSuccessfulTime, expectedExpense, reliability) =
  costExpectation iterationCount Projector workflow
let amortizedExpense =
  expectedExpense / reliability
let averageResponseTime =
  expectedSuccessfulTime / reliability
```

5.3.1 Loan request example

To evaluate the *cost* and *outcome* for the outermost Flow activity the algorithm computes *delayed costs* for the activities in the flow, and then sums them with the All compositor.

The first activity in the Flow is a Sequence of a Receive activity and an Assign activity for the *bigAmount* variable. There is no associated *cost* for the Receive activity, and the same holds for Assign. Also, their *outcome* is always Success. We thus compute the innerCost for the initial Sequence activity as $\text{Both}(\text{Zero}, \text{Delay}(\text{Zero}, \text{Zero})) = \text{Zero}$. Since there are no incoming `<link>`s the *delayedCost* is the same as the block's innerCost. We sample a value for *bigAmount* and store it into the *environment*. Let us assume that in this evaluation instance the variable *bigAmount* takes the True value. We also store in the *environment* the status of the outgoing `<link>`s, by evaluating the transition condition. In this case we set the *receiveToAssess* `<link>` to the left Sequence to False and the *receiveToApproval* `<link>` to the right Sequence to True.

Let us now consider the Sequence on the left. Since the status of its incoming `<link>` is false, its (implicit) *joinCondition* evaluates to false. Dead-path elimination is implemented by setting the output of the Sequence to Success, its *cost* to Zero, and the status of all its outgoing `<link>`s to False. The *delayedCost* for the Sequence is the Delay composition of the innerCost and the dependenciesCost. Since they are both Zero, also the *delayedCost* is Zero too.

The next activity is the Sequence on the right. Since the status of one of its incoming `<link>`s (the one with the top Sequence as source) is true, the *joinCondition* evaluates to true, and the Sequence is evaluated. First, we have to sample a value for the Invoke activity. Suppose the result of the first invocation yields a Fault, (0\$, 5 min). This will cause a new invocation (to a different endpoint, assigned by the fault handler). Suppose then that the second invocation yields a Success, (5\$, 10min), allowing the guard to be set to false and the loop to terminate, with Success outcome. The final innerCost for this block is equal to the *cost* for the loop (because the Assign has no cost), which is determined as follow:

```
Both((0$, 5 min), Delay((5$, 10 min), (0$, 5 min)))
= (5$, 15 min)
```

Note that this coincides with the *delayedCost* because both dependencies have Zero cost.

The bottom activity in the Flow is a Reply activity. There is no cost associated with this kind of activity, and hence innerCost = Zero. However, since the Reply depends upon other activities in the flow, viz., the assessment and approval Sequences, the *delayedCost* of the Reply activity is determined as:

TABLE 4: QoS estimation for different number of iterations for the loan request example.

Iteration Count	Reliability	Amortized Expense	Average Response Time
6	83%	9.06 \$	1200.6 sec
100	100%	6.80 \$	867.66 sec
10,000	99.3%	7.58 \$	940.75 sec
1,000,000	99.29%	7.60 \$	944.10 sec
100,000,000	99.30%	7.60 \$	944.51 sec
1,000,000,000	99.30%	7.60 \$	944.50 sec

$\text{Delay}(\text{Zero}, \text{All}(\text{Zero}, (5\$, 15 \text{ min}))) = (0\$, 15 \text{ min})$

The total cost for the `Flow` activity (and cost of the whole composition) is hence the `All` combination of `delayedCosts` of the activities inside it:

$\text{All}(\text{Zero}, \text{Zero}, (5\$, 15 \text{ min}), (0\$, 15 \text{ min})) = (5\$, 15 \text{ min})$

Table 3 summarizes the previously described trace along with other five runs of the `exec` function on the loan example.

By computing the above values for the samples of Table 3 we get:

$\text{expectedSuccessfulTime} = \frac{1}{6} \cdot (15 \cdot 60 + 2 + 181 + 25 \cdot 60 + 30 \cdot 60 + 0) = \frac{6003}{6} \text{ sec}$
 $\text{expectedExpense} = \frac{1}{6} \cdot (5 + 0.1 + 15.1 + 10 + 15 + 0.1) = \frac{45.3}{6} \$$
 $\text{reliability} = \frac{5}{6} = 83\%$
 $\text{amortizedExpense} = \frac{45.3}{5} = 9.06 \$$
 $\text{averageResponseTime} = \frac{6003}{5} = 1200.6 \text{ sec} = 20 \text{ min } 0.6 \text{ sec}$

In Table 4 we show how those QoS parameter estimations progressively converges by increasing the number of samples. The samples were generated using our implementation.

5.3.2 Shipping service example

A similar analysis can be performed for the shipping service example. In case the ship individually is `False` then the total cost of the whole workflow will be $\text{All}(\text{Zero}, (0.5\$, 2 \text{ sec}), \text{Zero}) = (0.5\$, 2 \text{ sec})$.

On the other hand, if ship individually is `True` and total items are three, then the loop will run three times and the inner cost of while loop will be $(1.5\$, 6 \text{ sec})$ which is also equal to the cost of the whole workflow. Table 5 summarizes the previously described trace along with other four runs of the `exec` function on the Shipping example. Reliability, expense and response time for shipping service example can be similarly estimated based on Table 5.

6 RELATED WORK

Various approaches (e.g., [5], [7], [8], [16], [17], [18], [19]) have been proposed to determine the QoS of service compositions.

Cardoso [8] presented a mathematical model and an algorithm to compute the QoS of a workflow composition. He iteratively reduces the workflow by removing parallel, sequence, alternative and looping structures according to a set of reduction rules, until only one activity remains. However, some workflow complex dependency structures cannot be decomposed into parallel or sequence, as shown in [7]. This kind of approach has been adopted also by other authors [17], [18], [19], some of whom (e.g., [16]) tried to reduce such limitation by defining more reduction patterns.

Mukherjee et al. [5], [7] presented a algorithm to estimate the QoS of WS-BPEL compositions. They convert a WS-BPEL workflow into an activity dependency graph, and assign probabilities of being executed to each activity. In their framework it is possible to treat any arbitrary complex dependency structure as well as *fault* driven flow control. However, [5], [7] do not consider correlations among activities which do not have a direct dependency, and this in some cases can yield a wrong result. For example, consider the case of diamond dependencies (mentioned in Section 2): four services A, B, C and D, such that B and C depend upon A, and D depends upon any of B or C being completed. If transition conditions from A to B and from A to C are mutually exclusive (viz., only one of them is satisfied at every execution) then exactly one among B and C is executed, and thus D is always executed (100% probability). However, [5], [7] consider the two conditions independent, allowing some probability of neither B nor C being executed. For example, if the transition conditions on B and C are true with 50% probability, there is a 25% probability of neither of them being executed, which yields a 75% probability of D being executed.

Zheng et al. [19] focused on QoS estimation for compositions represented by service graphs. They transform the service graph in order to remove the cycles, then calculate probabilities of execution and QoS parameters for each path. They claim that it is important not to aggregate results into a single number and to keep them as distributions to be able to identify problematic low probability cases which would be hidden by averages otherwise. For example, if a composition has response time of 10 ms 60% of times and of 100 ms 40% of times (average of 46 ms), it is not necessarily better than a composition that has 40 ms running time 60% of times and 60 ms running time 40% of times (average 48 ms). In their approach however they only marginally deal with parallelism, by not considering arbitrary synchronization `<link>s` (i.e., they restrict to cases in which is possible to decompose *flow*-like structures into parallel and sequences, as in [8]), and they do not take into account fault handling. Moreover, they need to fix an upper bound to the number of iterations for cycles, in order to allow decomposition into acyclic graph. They also assume that service invocations are deterministic, namely services are always successful and their QoS is not changing from one run to another.

Ivanovic et al. [20] defined a language to represent service compositions, and they address the problem of correlation. However the language does not describe parallel execution, thus their solution is similar to the ones proposed in workflow decomposition approaches ([8], [17], [18], [19]).

Moreover, to the best of our knowledge, all previous approaches require to know a priori the exact number of iterations, or at least an upper bound for each loop in order to estimate QoS values.

We conclude this section by summarising in Table 6 a qualitative comparison of our approach with respect to the two most related approaches i.e., Cardoso [8] and Mukherjee [5]¹³.

13. We opted for a qualitative comparison (w.r.t the challenges identified in Section 3), since other approaches require more information in the input and thus quantitative comparison was not possible. The legend for table data is:

TABLE 3: Total cost for different runs of the loan request example.

big Amount	high Risk	Risk Assessment	Approval(s)	Composition
True		Success (Zero)	Fault (0\$, 5 min);Success (5\$, 10 min)	Success (5\$, 15 min)
False	False	Success (0.1\$, 2 sec)		Success (0.1\$, 2 sec)
False	True	Success (0.1\$, 1 sec)	Success (15\$, 30 min)	Success (15.1\$, 181 sec)
True		Success (Zero)	Fault (0\$, 5 min);Success (10\$, 20 min)	Success (10\$, 25 min)
True		Success (Zero)	Success (15\$, 30 min)	Success (15\$, 30 min)
False		Stuck (0.1\$, 0)		Stuck (0.1\$, 0)

TABLE 5: Total cost for different runs of the shipping service example.

ShipIndividual	item < TotalItems	No of iteration	While loop	Composition
False	-	-	-	Success (0.5\$, 2 sec)
True	False	0	-	
True	True	1	Success (0.5\$, 2 sec)	Success (0.5\$, 2 sec)
True	True	2	Success (1\$, 4 sec)	Success (1\$, 4 sec)
True	True	3	Success (1.5\$, 6 sec)	Success (1.5\$, 6 sec)
...

TABLE 6: Qualitative comparison for the loan request and shipping service examples.

	Loan request example			Shipping service example		
	Cardoso [8]	Mukherjee [5]	Our algorithm	Cardoso [8]	Mukherjee [5]	Our algorithm
Correlation in parallel branches	N/A	N/A	N/A	Poor	Poor	Good
Non-determinism in the workflow	Poor	Poor	Good	Poor	Poor	Good
Different results of service invocations	Poor	Fair	Good	N/A	N/A	N/A
Complex Dependency structure	Poor	Good	Good	N/A	N/A	N/A

7 CONCLUDING REMARKS

We proposed a general approach probabilistically predict the QoS of service orchestrations. Our algorithm can tackle arbitrary dependency structures but it still preserves correlation, for example in diamond dependencies. Moreover, we proposed a simple model to calculate QoS for unbound loops, without imposing a fixed number of iterations. We also provided a more accurate handling of fault-driven control flow than previous approaches. Last, but not least, we accounted for the possibility that service invocations may not receive any response from the invoked service. A proof-of-concept implementation (called PASO) of the algorithm in F# was described.

While we presented our analysis on a subset of the WS-BPEL language, the same model can be applied to similar workflow languages. PASO can be fruitfully exploited both to probabilistically predict QoS values before defining the SLA of an orchestration and to compare the effect of substituting one or more endpoints (viz., remote services) in an orchestration.

We see different possible directions for future work. One of them is to extend our approach to model some other WS-BPEL constructs that we have not discussed in this paper, like *Pick* or *EventHandlers*. Another possible extension could be to deal with the case in which *no information at all* (not even a branch execution probability) is available for flow control structures. In order to identify the one with worst cost between two possible execution path, another composition function is needed. Similarly, the *uncorrelated samples* restriction imposed on invocations and assignments

- Good: Address the challenge.
- Fair: Partially address the challenge.
- Poor: Does not address the challenge.
- N/A: Not applicable for given example.

should be relaxed. We would also like to be able to specify some degree of correlation between consecutive samples (e.g., if a service invocations yields a fault because it is "down for maintenance" we should increase the probability of getting the same fault in the next invocation). Finally, the naive Monte Carlo implementation proposed (Section 3) is not the most efficient way to perform expectation queries on a distribution, and it can be improved in various ways [21], [22], [23], [24].

ACKNOWLEDGMENTS

This work was partly supported by the EU-FP7-ICT-610531 SeaClouds project.

REFERENCES

- [1] L. Bartoloni, A. Brogi, and A. Ibrahim, "Probabilistic prediction of the qos of service orchestrations: A truly compositional approach," in *Service-Oriented Computing*, ser. Lecture Notes in Computer Science, X. Franch, A. Ghose, G. Lewis, and S. Bhiri, Eds. Springer Berlin Heidelberg, 2014, vol. 8831, pp. 378–385. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-45391-9_27
- [2] M. Papazoglou, *Web services: principles and technology*, 2nd ed. Pearson Education Canada, Jan. 2012.
- [3] K. Kritikos, B. Pernici, P. Plebani, C. Capiello, M. Comuzzi, S. Benrenou, I. Brandic, A. Kertész, M. Parkin, and M. Carro, "A survey on service quality description," *ACM Computing Surveys (CSUR)*, vol. 46, no. 1, pp. 1–64, 2013.
- [4] D. Jordan, J. Evdemon, A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland *et al.*, "Web services business process execution language version 2.0," *OASIS standard*, vol. 11, Apr. 2007.
- [5] D. Mukherjee, P. Jalote, and M. G. Nanda, "Determining QoS of WS-BPEL compositions," in *Service-Oriented Computing-ICSOC 2008*. Springer, 2008, pp. 378–393.
- [6] W. L. Dunn and J. K. Shultis, *Exploring Monte Carlo Methods*. Elsevier, May 2011.
- [7] Debodoot Mukherjee, "QOS IN WS-BPEL PROCESSES," Master's thesis, Indian Institute Of Technology, Delhi, May 2008.

- [8] A. J. S. Cardoso, "Quality of service and semantic composition of workflows," PhD thesis, Univ. of Georgia, 2002.
- [9] D. Syme, A. Granicz, and A. Cisternino, *Expert F# 3.0*, 3rd ed. Berkeley, CA : New York: Apress, Oct. 2012.
- [10] T. H. Cormen, "Section 22.4: Topological sort," in *Introduction To Algorithms*. MIT Press, 2001, pp. 549–552.
- [11] A. Keller and H. Ludwig, "The WSLA framework: Specifying and monitoring service level agreements for web services," *Journal of Network and Systems Management*, vol. 11, no. 1, pp. 57–81, 2003.
- [12] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu, "Web services agreement specification (WS-Agreement)," in *Open Grid Forum*, vol. 128, 2007.
- [13] A. S. Vedamuthu, D. Orchard, F. Hirsch, M. Hondo, P. Yendluri, T. Boubez, and U. Yalcinalp, "Web services policy 1.5-framework," *W3C Recommendation*, vol. 4, pp. 1–41, 2007.
- [14] H. Becha, D. Amyot, and A. Boukerche, "Exposing and aggregating non-functional properties in SOA from the perspective of the service consumer," Ph.D. dissertation, University of Ottawa, 2012.
- [15] E. Kim, Y. Lee, Y. Kim, H. Park, J. Kim, B. Moon, J. Yun, and G. Kang, "Web services quality factors version 1.0," <http://docs.oasis-open.org/wsrm/WS-Quality-Factors/v1.0/cos01/WS-Quality-Factors-v1.0-cos01.html>, 2012.
- [16] M. Jaeger, G. Rojec-Goldmann, and G. Muhl, "QoS aggregation for web service composition using workflow patterns," in *Enterprise Distributed Object Computing Conference, 2004. EDOC 2004. Proceedings. Eighth IEEE International*, Sep. 2004, pp. 149–159.
- [17] N. B. Mabrouk, S. Beauche, E. Kuznetsova, N. Georgantas, and V. Issarny, "QoS-Aware service composition in dynamic service oriented environments," in *Middleware 2009*, ser. Lecture Notes in Computer Science, J. M. Bacon and B. F. Cooper, Eds. Springer Berlin Heidelberg, Jan. 2009, no. 5896, pp. 123–142. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-642-10445-9_7
- [18] H. Wang, H. Sun, and Q. Yu, "Reliable service composition via automatic QoS prediction," in *2013 IEEE International Conference on Services Computing (SCC)*, Jun. 2013, pp. 200–207.
- [19] H. Zheng, W. Zhao, J. Yang, and A. Bouguettaya, "Qos analysis for web service compositions with complex structures," *Services Computing, IEEE Transactions on*, vol. 6, no. 3, pp. 373–386, 2013.
- [20] D. Ivanovic, P. Kaowichakorn, and M. Carro, "Towards qos prediction based on composition structure analysis and probabilistic environment models," in *Principles of Engineering Service-Oriented Systems (PESOS), 2013 ICSE Workshop on*. IEEE, 2013, pp. 11–20.
- [21] S. Bhat, J. Borgström, A. D. Gordon, and C. Russo, "Deriving probability density functions from probabilistic functional programs," in *19th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2013, eAPLS Best Paper Award for ETAPS 2013. [Online]. Available: <http://research.microsoft.com/apps/pubs/default.aspx?id=189021>
- [22] B. Milch, L. S. Zettlemoyer, K. Kersting, M. Haimes, and L. P. Kaelbling, "Lifted probabilistic inference with counting formulas," in *Proc. 23rd AAAI Conference on Artificial Intelligence*, 2008, pp. 1062–1068.
- [23] N. Ramsey and A. Pfeffer, "Stochastic lambda calculus and monads of probability distributions," in *ACM SIGPLAN Notices*, vol. 37, 2002, pp. 154–165. [Online]. Available: <http://dl.acm.org/citation.cfm?id=503288>
- [24] A. Stuhlmüller and N. D. Goodman, "A dynamic programming algorithm for inference in recursive probabilistic programs," in *Second Statistical Relational AI workshop at UAI 2012 (StaRAI-12)*, 2012.